Posted September 11, 2010 by [Eric Niebler](http://ericniebler.com), under [Boost](http://cpp-next.com/archive/category/boost/)

From <http://cpp-next.com/archive/2010/08/expressive-c-introduction/>

**Expressive C++: Playing with Syntax**

This entry is part of a series, [Expressive C++»](javascript:;) Entries in this series:

1. [Expressive C++: Introduction](http://cpp-next.com/archive/2010/08/expressive-c-introduction/)
2. **Expressive C++: Playing with Syntax**
3. [Expressive C++: Why Template Errors Suck and What You Can Do About It](http://cpp-next.com/archive/2010/09/expressive-c-why-template-errors-suck-and-what-you-can-do-about-it/)
4. [Expressive C++: A Lambda Library in 30 Lines (Part 1 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-one/)
5. [Expressive C++: A Lambda Library in 30 Lines (Part 2 of 2)](http://cpp-next.com/archive/2010/10/expressive-c-expression-extension-part-two/)
6. [Expressive C++: Fun With Function Composition](http://cpp-next.com/archive/2010/11/expressive-c-fun-with-function-composition/)
7. [Expressive C++: Trouble With Tuples](http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/)
8. [Expressive C++: Expression Optimization](http://cpp-next.com/archive/2011/01/expressive-c-expression-optimization/)

Welcome to the second in a series of articles devoted[[1]](#footnote-1) Domain-Specific Embedded Languages in C++. In the [intro](http://cpp-next.com/archive/2010/08/expressive-c-introduction/) I discussed the benefits of DSELs (“dee-sells”), and talked up[[2]](#footnote-2) the possibilities offered by Boost.Proto, a DSEL compiler-construction toolkit. I also threw in[[3]](#footnote-3) examples like Boost.Spirit, a large and complex DSEL that approximates EBNF in C++.

After all that talk[[4]](#footnote-4), you might be forgiven for thinking that[[5]](#footnote-5) all DSELs are big and complicated, and that this whole DSEL thing is not very practical. Not so. In this article, I’ll step through the design and implementation of a fairly simple DSEL[[6]](#footnote-6). By the end, you should be able to write your own small DSELs that are actually useful.

**Mad Libs**[**1**](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fn:1) **Formatting**

Here’s a little problem. You have a [Mad Libs](http://www.madlibs.com/)-like string formatting API that takes a format string like "There are more things in {place}, Horatio, than are dreamt of in your {thing}.\n" that contains placeholders, and a std::map containing substitutions.

// Declare and initialize a map containing substitutions

std::map<std::string, std::string> subst;

subst["place"] = "heaven and earth";

subst["thing"] = "philosophy";

// Prints: There are more things in heaven and earth, Horatio,

// than are dreamt of in your philosophy.[[7]](#footnote-7)

std::cout << format( "There are more things in {place}, Horatio, "

"than are dreamt of in your {thing}.**\n**", subst );

Calling this API is a pain because declaring and filling a std::map is a multi-line proposition[[8]](#footnote-8). Recognizing that laziness is one of the traits of a great programmer[**2**](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fn:2), you’d like to change this multi-liner into a one-liner. It’s easily done, and the best part is: you get to design your own language.

**DSEL Language Design**

Now comes the fun part[[9]](#footnote-9): getting to decide what your mini-language looks like[[10]](#footnote-10). The first step is to understand the peculiar rules of C++ expression-based DSELs[[11]](#footnote-11): your language must consist of valid C++ expressions. You can only use C++ identifiers, functions and operators, and your language must obey the precedence and associativity rules of C++. So for instance, as nice as it may be, this is right out[[12]](#footnote-12):

// WRONG: not a valid C++ expression

format( "There are...", {"place": "heaven and earth",

"thing": "philosophy"} );

These however, are valid and reasonable approaches (for some appropriately defined map object):

// OK, valid C++ expressions:

format( "There are...", map("place", "heaven and earth")

("thing", "philosophy") );

format( "There are... ", map ["place"]("heaven and earth")

["thing"]("philosophy") );

format( "There are...", map["place"] = "heaven and earth" +

map["thing"] = "philosophy" );

format( "There are...", ( map["place"] = "heaven and earth",

map["thing"] = "philosophy") );

Notice how in each case, we’ve used expressions that obey the syntax of the C++ language. Take a moment to think about the operator overloads that would make each expression compile. For instance, the first expression requires the map object to have an overloaded function call operator that takes two arguments and returns something that itself has a function call operator, etc.[[13]](#footnote-13)

**Gotcha!** Be aware that you can’t assign a domain-specific meaning to an expression that already means something in standard C++. Consider:

map ("place" == "heaven and earth")

("thing" == "philosophy")

In C++, applying the equality operator on two string literals ends up comparing two pointers. Oops. When crafting your DSEL, you need to be careful not to set *your users up for this kind of a fall.13*

The last one with the overloaded comma operator is interesting. Notice how I had to wrap the expression in an extra set of parentheses. The comma’s dual nature as an operator and as a separator in argument lists makes it persnickety to work with in DSELs. **For Extra Credit[[14]](#footnote-14)** :One of these expressions is not like the others. Think about operator precedence. Can you spot the odd man out[[15]](#footnote-15)?

**Operators Are Standing By[[16]](#footnote-16)**

Perhaps you’re thinking, “How will I even get that to *compile?*” That’s actually the easy part, and it’s where Proto shines. With just the following, the above code compiles and runs (and does nothing):

#include <string>

#include <boost/proto/proto.hpp>

struct map\_ {};

boost::proto::terminal<map\_>::type map = {};

template< class Expr >

void format( std::string fmt, Expr const & expr )

{}

int main()

{

// OK, valid C++ expressions:

format( "There are...", map ("place", "heaven and earth")

("thing", "philosophy") );

format( "There are...", map["place"]("heaven and earth")

["thing"]("philosophy") );

format( "There are...", map["place"] = "heaven and earth" +

map["thing"] = "philosophy" );

format( "There are...", (map["place"] = "heaven and earth",

map["thing"] = "philosophy") );

}

We included a header, defined a symbol and *that’s it*. Proto defines operator overloads so that any Proto expression, like the map terminal above, can be combined into larger Proto expression trees. You get that for free.

**Heterogeneous Trees**

The expression trees Proto builds are not what a Java programmer or an OO advocate might expect. There are no dynamic allocations, no abstract base classes, and no virtual member functions. It is a *heterogeneous* tree; each node has full static type information about what operation created it and what the types of its child nodes are. It is also built without any copying, which makes it essentially free at runtime (assuming the optimizer is doing its job).

**Visualizing Expression Trees**

Let’s take a closer look at two of the expressions above and view the structure of the tree Proto builds for us. Here are the trees for the last two expressions, the one with the comma operator, and the one with plus:

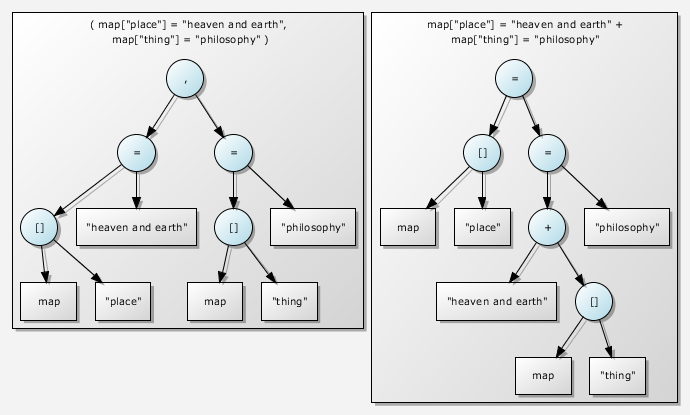
[](http://cpp-next.com/wp-content/uploads/2010/08/02-fig1.dot_.notugly2.png)

Figure 1: Comparison of two expression trees

Wow, what a different an operator makes![[17]](#footnote-17) The expression with the comma gives a very sensible tree where each placeholder is near its associated substitution string. That will make the tree easier to interpret as a map later. The expression with the plus has a totally different structure where the placeholder "place" is nowhere near its substitution string "heaven and earth" (which is what you’d need to move to make any use of this expression). The fact that the substitution node doesn’t have a simple relative location to its associated placeholder would make it very difficult to interpret this tree as a map.  
**Kudos[[18]](#footnote-18)** : If you took the “Extra Credit” quiz above and correctly guessed that the expression with the plus was not like the others, give yourself a pat on the back[[19]](#footnote-19). Because of its structure, it’s largely useless for our purposes.

The reason for the difference is the precedence of the + operator. It has higher precedence than the = operator, so that (a = b + c = d) groups as (a = (b + c) = d). The comma operator has very low precedence, so (a = b, c = d) groups as ((a = b), (c = d)). That’s more like what we want. The lesson is that the rules of C++ are fixed and immutable; ignore them at your own peril![[20]](#footnote-20)

Boost.Proto gives you a handy tool for exploring the structure of expressions. Just pass them to display\_expr to pretty-print them and get a view of their structure. For instance, this:

proto::display\_expr( (map["place"]="heaven and earth", map["thing"]="philosophy") );

writes the following to std::cout[**3**](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fn:3):

comma(

assign(

subscript(

terminal(struct map\_)

, terminal(place)

)

, terminal(heaven and earth)

)

, assign(

subscript(

terminal(struct map\_)

, terminal(thing)

)

, terminal(philosophy)

)

)

The upshot[[21]](#footnote-21) is that Boost.Proto makes it painless to explore the design space for your domain-specific language. Experiment and have fun, commitment-free.

**Walk a Tree, Build a Map**

We’ve explored our design space, and it’s time to settle on[[22]](#footnote-22) a syntax for our little language. Since the goal of this exercise was to save typing, let’s go with the most concise syntax:

format( "There are...", map ("place", "heaven and earth")

("thing", "philosophy") );

If we pass the map expression to proto::display\_expr, we see something like this:

function(

function(

terminal(struct map\_)

, terminal(place)

, terminal(heaven and earth)

)

, terminal(thing)

, terminal(philosophy)

)

That’s really a very simple tree; there are only two non-terminal nodes and they’re both function call invocations. These nodes each have three child nodes: the “function”, the key, and the value. The “function” in this case is either the map terminal or another function node, making this a recursive data structure.

The natural way to process a recursive data structure is with recursive functions. We can easily walk this tree and fill in a std::map as we go[[23]](#footnote-23). The following two functions do just that. It is explained below:

typedef std::map<std::string, std::string> string\_map;

// Recursive function used to fill the map

template< class Expr >

void fill\_map( Expr const & expr, string\_map & subs )

{

using boost::proto::value; // read a value from a terminal

using boost::proto::child\_c; // get the Nth child of a non-terminal

subs[ value(child\_c<1>(expr)) ] = value(child\_c<2>(expr));

fill\_map( child\_c<0>(expr), subs );

}

// The 'map' terminal ends the recursion

void fill\_map( boost::proto::terminal<map\_>::type const &, string\_map & )

{}

We’ll use fill\_map by passing it both the expression tree and a std::map to fill. It works by peeling off one layer of the tree at a time, using the two terminals at each layer to insert a new key/value pair into the map.

Tree Traversals

For simple trees, writing your own recursive functions to process them is not that difficult. For trees with richer structure, however, this approach quickly becomes impractical. To bring this discussion back around to compiler construction, it would be as if yacc built an AST for you and then walked off the job, leaving you to walk the AST and generate code by yourself. Rest assured that Proto offers a more powerful way to traverse and manipulate expression trees that will be described in future articles.

fill\_map introduces two of Proto’s accessor functions: value and child\_c. value is used to access the value in a Proto terminal node. child\_c<N> retreives the Nth child of a non-terminal.

**Putting the Pieces Together**

Now that we have a way to walk the tree and build a std::map, the rest is cake.[[24]](#footnote-24) We simply offer a new format API that forwards to the old one and voila! we’re done[[25]](#footnote-25). For those keeping score[[26]](#footnote-26), the completed solution looks like this:

////////////////////////////////////////////////////////////////////////////

//

// A simple DSEL for creating map-like associations

//

////////////////////////////////////////////////////////////////////////////

#include <map>

#include <string>

#include <iostream>

#include <boost/proto/proto.hpp>

#include <boost/xpressive/xpressive.hpp>

#include <boost/xpressive/regex\_actions.hpp>

struct map\_ {};

boost::proto::terminal<map\_>::type map = {};

typedef std::map<std::string, std::string> string\_map;

// Recursive function used to fill the map

template< class Expr >

void fill\_map( Expr const & expr, string\_map & subs )

{

using boost::proto::value; // read a value from a terminal

using boost::proto::child\_c; // get the Nth child of a non-terminal

subs[ value(child\_c<1>( expr )) ] = value(child\_c<2>(expr));

fill\_map( child\_c<0>(expr), subs );

}

// The 'map' terminal ends the recursion

void fill\_map( boost::proto::terminal<map\_>::type const &, string\_map & )

{}

// The old format API that accepts a map of string substitutions

std::string format( std::string fmt, string\_map & subs )

{

namespace xp = boost::xpressive;

using namespace xp;

sregex const rx = '{' >> (s1= +\_w) >> '}'; // like "{(\\w+)}"

return regex\_replace(fmt, rx, xp::ref(subs)[s1]);

}

// The new format API that forwards to the old one

template< class Expr >

std::string format( std::string fmt, Expr const & expr )

{

string\_map subs;

fill\_map( expr, subs );

return format( fmt, subs );

}

int main()

{

std::cout << format("There are more things in {place}, Horatio, "

"than are dreamt of in your {thing}.\n"

, map("place", "heaven and earth")

("thing", "philosophy") );

}

As expected, the code prints out:

There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy.

**What’s boost::xpressive ?**

I snuck[[27]](#footnote-27) some extra code in there to see if you were paying attention. It’s [Boost.Xpressive](http://www.boost.org/libs/xpressive), a regular expression and string manipulation library. The first format overload[[28]](#footnote-28) builds a regular expression:

sregex const rx = '{' >> (s1= +\_w) >> '}'; // like "{(\\w+)}"

The next line passes the regex to regex\_replace to actually do the substitution:

return regex\_replace(fmt, rx, xp::ref(subs)[s1]);

The third parameter is a kind of anonymous substitution function. It says: take the part of the string that matched the pattern +\_w (a whole word), use it to index into the subs map, and use the result as the substitution string. xp::ref is needed to delay the evaluation of the index operation until after the match has occurred. The regex and the anonymous function are from two different DSELs, but notice how they collaborate with each other through the shared use of the s1 placeholder. This kind of DSEL composition is one of the most powerful and exciting features of DSELs.

Counting the DSEL we just defined, this source code uses a total of 3 DSELs. They’re all built with Proto.

**Conclusions and What’s to Come**

In this article, we built a little DSEL for expressing map-like relationships inline. With the help of Proto, it was pretty easy. In fact, one could argue that the term “DSEL” doesn’t even apply to such a simple library interface. It’s true: the boundary between DSELs and merely clever library interfaces is a fuzzy one. But even for the most borderline cases[[29]](#footnote-29) like this, Proto comes with a collection of tools[[30]](#footnote-30) to make the job easier.

In the next article, we’ll firm up[[31]](#footnote-31) the specification of this library interface, add some parameter checking, and use some nifty techniques[[32]](#footnote-32) to improve template error reporting.

Until next time!

1. Mad Libs is a registered trademark of Penguin Group (USA) Inc. [↩](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fnref:1)
2. “We will encourage you to develop the three great virtues of a programmer: laziness, impatience[[33]](#footnote-33), and hubris[[34]](#footnote-34).”, Larry Wall, Programming Perl (1st edition), O’Reilly And Associates. [↩](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fnref:2)
3. Tested on Visual C++ 10 2010 [↩](http://cpp-next.com/archive/2010/09/expressive-c-playing-with-syntax/#fnref:3)

Posted Saturday, September 11th, 2010 under [Boost](http://cpp-next.com/archive/category/boost/).

1. a series of articles devoted ~ : ~를 다루고 있는 연재물. ex) a book devoted to Korean history :한국 역사를 다룬 책 [↑](#footnote-ref-1)
2. talk up : 있어 보이게 얘기하다. to speak of or discuss favorably in order to arouse interest or support : [↑](#footnote-ref-2)
3. throw in : 덤으로 주다. [VERB] to add (something extra) at no additional cost. [↑](#footnote-ref-3)
4. After all that talk : 다 듣고 나서는, [↑](#footnote-ref-4)
5. might be forgiven for thinking that ~ : ~ 라 생각할 수도 있다. [↑](#footnote-ref-5)
6. step through the design and implementation of a fairly simple DSEL : 아주 간단한 DSEL 의 설계와 구현 전체 과정을 밟아 가다 [↑](#footnote-ref-6)
7. There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy : 호레이쇼, 이 천지간에는 말일세, 자네의 철학으로는 상상도 못할 일이 너무나 많다네. - Shakespeare 햄릿 1막 5장 - [↑](#footnote-ref-7)
8. a multi-line proposition : 여러 줄에 걸친 것 [↑](#footnote-ref-8)
9. Now comes the fun part : 이제 재미있는 일이 생긴다. [↑](#footnote-ref-9)
10. getting to decide what your mini-language looks like : 우리의 '작은 언어'가 무엇처럼 보이기를 정하는 일 말이다. [↑](#footnote-ref-10)
11. the peculiar rules of C++ expression-based DSELs :C++ 표현식에 기반한 DSEL들의 좀 이상한 규칙들 [↑](#footnote-ref-11)
12. as nice as it may be, this is (right) out : 멋져 보일 수는 있으나 이런 사용법은 (절대로) 허용되지 않는다. [↑](#footnote-ref-12)
13. set your users up for this kind of a fall : 이와 같은 재앙을 사용자들에게 제공하다 [↑](#footnote-ref-13)
14. Extra Credit : 추가 점수. [↑](#footnote-ref-14)
15. Can you spot the odd man out ? : 그 독특한 녀석을 구분해 낼 수 있겠는가?  
     the odd man[one] out : a person or thing that is different from others or does not fit easily into a group or set [↑](#footnote-ref-15)
16. Operators are standing by : 연산자들은 이미 준비되어 있다. [↑](#footnote-ref-16)
17. Wow, what a different an operator makes! : 세상에! 한 개의 연산자가 만들어 낸 다름을 보라. [↑](#footnote-ref-17)
18. Kudos : 칭찬 받기. 잘 했어요 [↑](#footnote-ref-18)
19. give oneself a pat on the back : 자화자찬하다 [↑](#footnote-ref-19)
20. at your (own) peril : 아주 위험하게 [↑](#footnote-ref-20)
21. upshot : 결론.   
     the upshot of a series of events or discussions is the final result of them, usually a surprising result. [↑](#footnote-ref-21)
22. settle on ~ : ~ 을 정하다. [↑](#footnote-ref-22)
23. We can easily walk this tree and fill in a std::map as we go : 쉽게 이 트리를 순회 할 수 있고, 그러면서 맵을 채울 수 있다. [↑](#footnote-ref-23)
24. the rest is cake : 나머지는 껌이다.

    [PHRASE][usu v-link PHR] If you think something is very easy to do, you can say it is a piece of cake.   
     People often say this to stop someone feeling worried about doing something they have to do. [↑](#footnote-ref-24)
25. voila! we're done : 보고 있나! 우린 해 냈다. [↑](#footnote-ref-25)
26. for those keeping score : ? 챙겨 보면, 정리해 보면,... [↑](#footnote-ref-26)
27. sneak : 몰래 하다 [↑](#footnote-ref-27)
28. the first format overloaded... : 첫 번째 오버로딩 된 format 함수는... [↑](#footnote-ref-28)
29. a borderline case 이것도 저것도 아닌 [애매한] 경우 [↑](#footnote-ref-29)
30. Proto comes with a collection of tools...: Proto 는 도구 모음을 제공한다[가지고 있다]. [↑](#footnote-ref-30)
31. firm up : 확정하다 [↑](#footnote-ref-31)
32. nifty technique : 세련된 기법들. If you describe something as nifty, you think it is neat and pleasing or cleverly done. [↑](#footnote-ref-32)
33. impatience : 조급함. (…하고 싶어) 못 견딤. 초조함. [↑](#footnote-ref-33)
34. hubris : 자만심 [↑](#footnote-ref-34)